

第6章

トランジスタ構成まで意識した演算回路の最適化設計

――ゲートの内部構造を理解して設計の幅を広げよう

佐藤 証

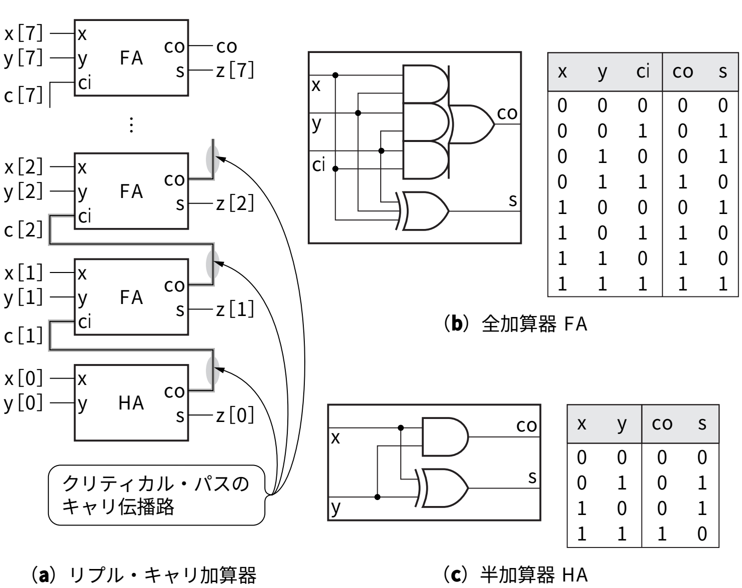
ここでは，ゲート・レベルよりさらに掘り下げて，トランジスタ・レベルの回路構成を考慮した最適化の事例を紹介する．大規模な回路の場合，人手によるライブラリ・マッピングで性能を向上できる余地がかなりある．こうした最適化を行うには，データシートを熟読し，ゲートの内部構造やトランジスタの特性などを押さえることが肝要と筆者は言う．　（編集部）

ひと昔前は，加算器を作るだけでも「ハーフ・アダーとフル・アダーのセルをこう組み合わせて，ここにキャリを飛ばせて…」などとあれこれ考えながら設計していました．一方，現在では，

```
assign z = x + y;
```

と1行書けば，後は論理合成ツールが制約条件に合わせてネットリストを作ってくれます．さらにC言語ベースの回路設計が普及しはじめている今日，筆者の得意とするトランジスタのマスク・パターンを直接描くカスタム・レイアウトのスキルは，きわめて特殊な場合にしか必要とされないのかもしれません．

しかし，マイクロプロセッサのアーキテクチャやアセンブリ言語に詳しい人が質の高いプログラムを書くように，ASICライブラリのトランジスタ構成や回路特性に関する知識があると，RTLでも高性能な回路を設計できます．とくに，算術演算回路はアーキテクチャの自由度が高いため，どれだけ多くのテクニックが使えるかによって性能に大きな差が出てきます．



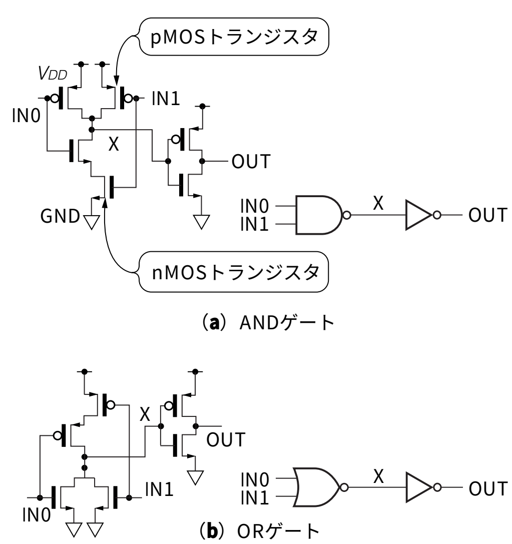
x	y	ci	co	s
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

x	y	co	s
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

（a）リプル・キャリ加算器　（b）全加算器 FA　（c）半加算器 HA

図1　リプル・キャリ加算器の構成

(a)の下方の加算器（FAまたはHA）からのキャリがAND-ORゲートを通って上のFAへと順々に伝播していく．



（a）ANDゲート　（b）ORゲート

図2　2入力ANDゲートとORゲートのトランジスタ構成

ANDゲートやORゲートは，NAND-INVERTERやNOR-INVERTERといったように二つのゲートがシリアルに接続されていることがわかる．

そこで，ここでは「回路設計テクニックの幅を広げる」という意味で，簡単な演算回路を例にトランジスタ構成を意識した人手によるライブラリ・マッピング「手マップ」の技を紹介したいと思います．

1 リプル・キャリ加算器の最適化

図1(a)は論理回路の入門書にもよく出てくるリプル・キャリ加算器です．8ビットの整数x[7：0]とy[7：0]を入力すると，8ビットの整数z[7：0]と1ビットのキャリcoを加算結果として出力します．全加算器は半加算器2個で記述されることも多いのですが，ここでは多入力のXORとAND-ORゲートを使っています．下の全加算器FA（あるいは半加算器HA）からやってきたキャリは，AND-ORゲートを通って上のFAへと順々に伝播していきます．その名は，このキャリの伝播が「さざ波(ripple)」を連想させることに由来しています．

リプル・キャリ加算器はシンプルで小型である半面，キャリが伝播するクリティカル・パスは演算ビット数に比例して長くなるため，とても低速です．**図1**のキャリ伝播のパスには2入力ANDゲートが一つと，6入力AND-ORゲートが七つ入っています．高速な演算にはキャリ・ルックアヘッド加算器などが用いられますが，ここではあくまでも例題ということで，リプル・キャリ加算器の高速化を考えてみましょう．

HDLによるリプル・キャリ加算器のRTL記述は，XOR，AND，ORの三つの演算子を用いて**図1**のとおりに作るのが正解です．しかし，トランジスタ・レベルの設計者の視点では，この回路は不合格です．CMOSライブラリの基本ゲートはほとんどが反転出力を持ちます．直接コンポーネントを指定できるのであれば，それらのゲートにNANDやAND-OR-INVERTER(AOI)などを使うことで，速度と規模の双方で回路を最適化できます．例として，2入力のANDゲートとORゲートのトランジスタ構成を**図2**に示します．これから，CMOSのANDゲートやORゲートはNAND-INVERTERやNOR-INVERTERといったように二つのゲートがシリアルに接続されていることがわかります．

そこで，**図1**のクリティカル・パスのANDとAND-ORを**図3**のようにそれぞれNANDとAOIに置き換え，キャリの極性を負論理・正論理と交互にすれば，八つ分のインバータ遅延を削減することができます．

● 多入力ゲートの速い入力ポートと遅い入力ポート

次にライブラリの各ゲートの内部構造を考慮した高速化テクニックを紹介しましょう．**図2**のANDゲートとORゲートの各入力は論理面では等価なのですが，よく見るとトランジスタ間の配線は異なっています．このため，入力キャパシタンスや出力までの遅延時間に差が出てきます．で

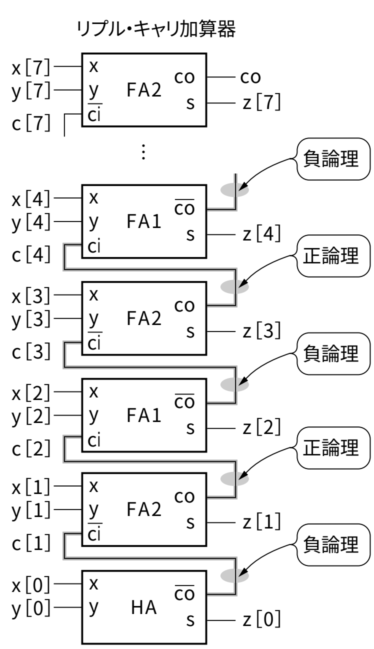
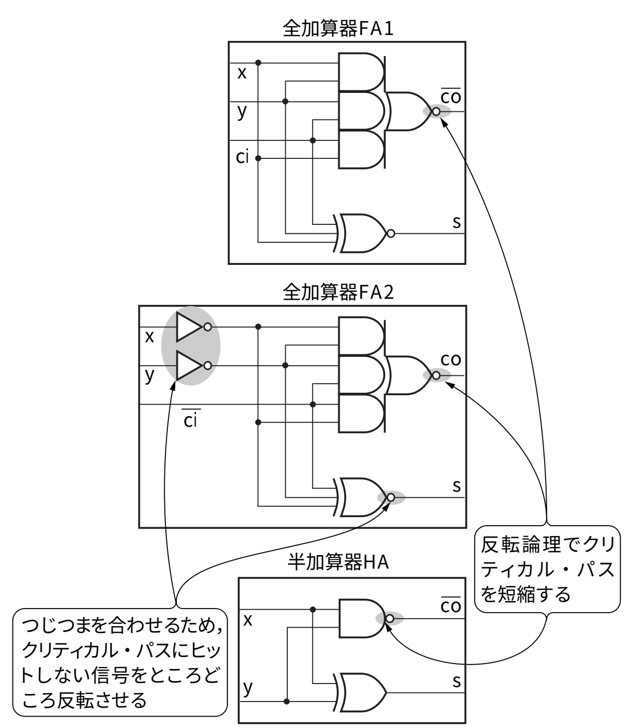


図3 高速（？）リプル・キャリ加算器の構成
図1でクリティカル・パスとなったANDとAND-ORを，それぞれNANDとAOIに置き換えた．



はNANDゲートを例に，2入力IN0とIN1の一方が速く動作する理由をトランジスタ・レベルで説明しましょう．

図4(a)と**(b)**は，2入力(IN0，IN1)が以下のように変化した場合の二つのパターンを示しています．

(a) (0，0)→(0，1)→(1，1)
(b) (0，0)→(1，0)→(1，1)

‘0’はGND，‘1’はV_{DD}電位とします．V_{DD}側にあるpMOSトランジスタはゲート入力が‘0’になるとON状態となり，信号‘1’を通します．また，GND側にあるnMOSトランジスタはゲート入力が‘1’になると信号‘0’を通す特性を持っています．nMOSがシリアルにつながっている途中のノードNは，初期状態でフローティングになりますが，‘1’の状態にあるものと仮定します．

図4(a)では，IN1が‘0’→‘1’と遷移することで，右側のpMOSトランジスタTP1がOFFに，下側のnMOSトランジスタTN1がONになります．TP0がONのままなので出力OUTは‘1’を保持し，またノードNはGNDにつながって‘1’→‘0’と遷移します．次にIN0が‘0’→‘1’と遷移するとTP0がOFFとなり，V_{DD}からOUTへのパスが切れ，逆にTN0がONすることでGNDへのパスができ，OUTは‘1’→‘0’と変化します．

図4(b)ではIN0が先に‘0’→‘1’と変わり，TP0がOFF，TN0がONとなります．出力OUTは**図4(a)**と同じく‘1’のままですが，ノードNはGNDではなく逆にV_{DD}につながるので‘1’を保持します．次にIN1が‘0’→‘1’と遷移すると，TP1がOFFし，TN1がONして，まずノードNが‘1’→‘0’となり，それに続いてOUTが‘1’→‘0’と変化します．

図4(a)ではnMOSが両方ONしたときに，GND電位が出力OUTのすぐ近くまできていますが，**図4(b)**では二つのnMOSを順番に通ってOUTが‘0’に引き抜かれるため，遅くなっていることがわかります．つまり，多入力ゲートで論理的に等価な入力ポートがあれば，遅れてくる信号をできるだけ出力ポートに近いトランジスタのゲートにつなぐというのがここでのポイントです．

図5に論理的に等価な3入力を持つゲートの例をいくつか示します．いずれのゲートもいちばん遅くやってくる信号はIN0に，そしていちばん早い信号はIN2につなぐのが正解です．AOIゲートは少しわかりづらいかもしれませんが，自分で**図4**のようにトランジスタのON/OFFの過程を追ってみると，ここで紹介したテクニックへの理解が深まると思います．

● ライブラリのデータ・ブックを熟読する

さて，反転論理の使用や入力ポートの選びかたが理解できたところで，実際にはどうコーディングすればよいのでしょう．

```
assign OUT = ~(IN0 & IN1 & IN2);
```

と書いても，論理合成ツールが3入力NANDゲートにマップする保証はありません．また，たとえそうであったとしても入力ポートの接続がどうなるのかわかりません．そこで，ASICライブラリのデータ・ブックで必要な論理ゲー

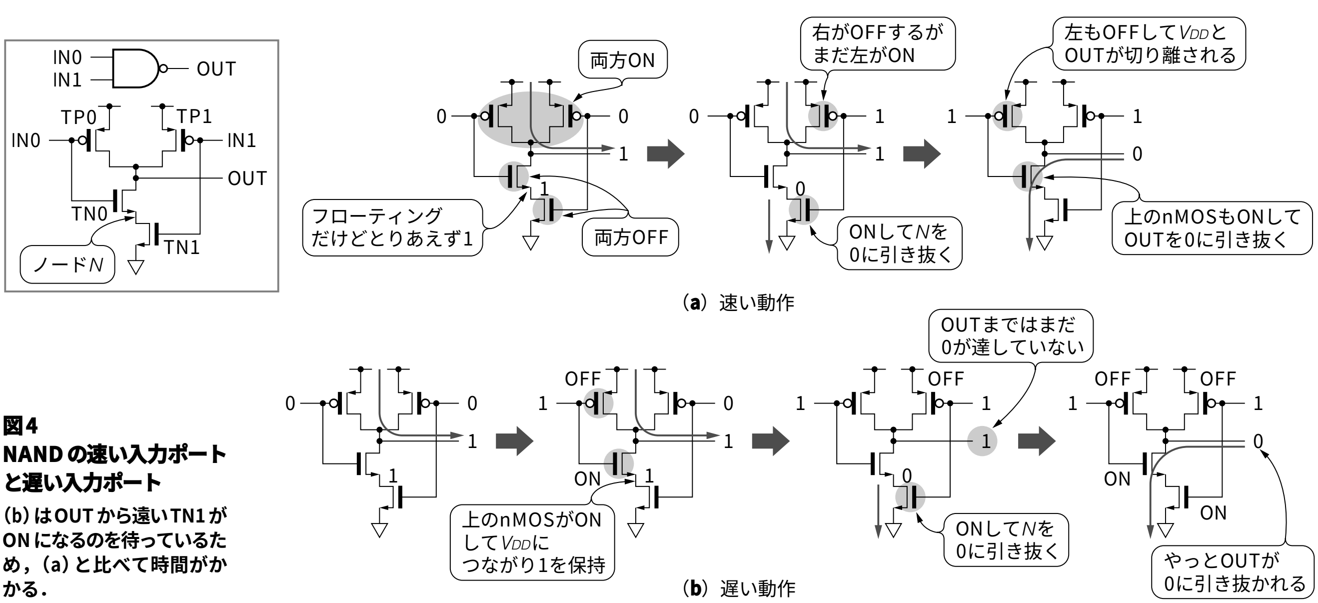


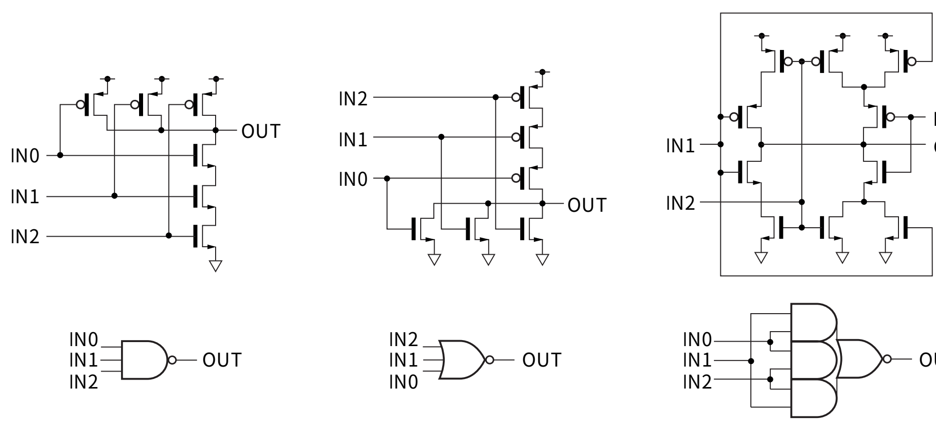
図4
NANDの速い入力ポートと遅い入力ポート
(b)はOUTから遠いTN1がONになるのを待っているため，(a)と比べて時間がかかる．

トの名まえとポートの配置を調べ，HDLコードでそのコンポーネントを直接指定します．これを筆者の周りでは「手マップ」と呼んでいます．もちろん論理合成ツールによって「手マップ」した部分をいじられないようにオプション設定することも忘れてはなりません．

データ・ブックには各ゲートの入出力ポートの‘0’→‘1’，‘1’→‘0’といった遷移方向に応じた遅延時間や，入力キャパシタンス，そして回路面積などの情報が載っていますが，内部のトランジスタ構成まで書かれていることはほとんどありません．カスタム・レイアウトのしごとに携わっていると，これらのデータからゲート内部の構造がわかるようになりますが，そこまで知らなくても遅延時間やキャパシタンスの情報だけでもかなりの最適化が可能です．そもそも，論理合成ツールはこれらの数値を使って最適化しているわけですし…（だから手マップにはかなわない？）．

CMOS回路で1ゲートという場合，通常は**図4**の四つのトランジスタのNANDゲートを指します．しかし，NANDゲートと言っても1種類ではありません．**図6**のように，パワー・レベル（駆動能力）の違いによってトランジスタ数が異なるものが複数用意されています．十数ものバリエーションを持つゲート論理も珍しくありません．

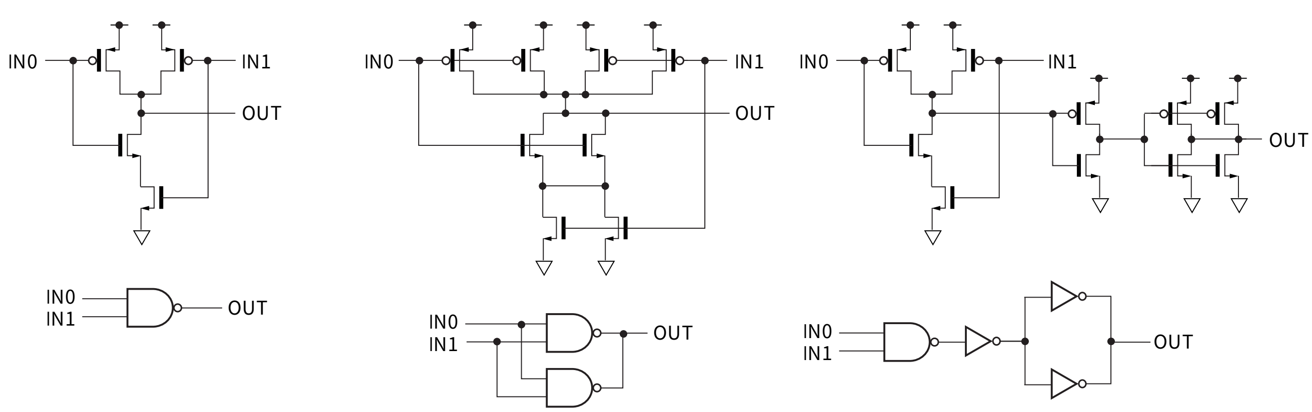
これらを，負荷容量の小さい場所には小さなゲートを，大きい場所には大きなゲートを，というように適切に使い分ける必要があります．負荷の大きいところに小さなゲートを置くと駆動に時間がかかってしまいます．逆に負荷の小さい場所に内部遅延時間の長い大きなゲートを使うのも逆効果です．手マップの場合，どこにどの程度のパワー・レベルのゲートが必要なのかをすべて正確に予測することは困難です．しかし，算術演算回路は，キャリなどのごく一部の信号を除いて配線がローカルであったり，構造が規則的であったりするので，それほどファンアウト調整に手こずることはないと思います．実際の作業としては，すべてのゲートを最小のものにマップして，遅延レポートで遅



（a）3入力NANDゲート　（b）3入力NORゲート　（c）3入力AOIゲート

図5 ゲートを速く動作させる入力ポートのつなぎかた

いずれのゲートも三つの入力ポートは論理的に等価だが，GNDあるいはV_{DD}から出力OUTにシリアルにつながれたトランジスタが順序良くONするように信号をつなぐと，高速に動作する．



（a）パワー・レベル小　（b）パワー・レベル中　（c）パワー・レベル大

図6 NANDゲートのバリエーション

論理的には一つしかない2入力NANDゲートにも多くのバリエーションが存在し，パワー・レベルによってゲートを構成するトランジスタの数が異なる．

い信号に対するゲートを大きくしたり，バッファを挿入するといった作業を数回繰り返します．慣れないうちは，あるゲートを大きくするとそのゲートの駆動に大きなゲートが必要になったり，またあるクリティカル・パスを短くすると別の経路がクリティカル・パスとして現れたりと，モグラ叩きの状態になるかもしれません．このあたりは勘を養うことが重要なのですが，それにはゲート内部のトランジスタ構成を考えることも必要です．

データ・ブックを眺めていると，使い道のよくわからないへんてこな複合ゲートに出会うことがあります．しかし，論理式の項数を最小にしたからといって，そのまままきれいにライブラリにマップされるとは限らず，へんてこなゲートを使ったほうがむだのない場合もあります．ですから，手マップをマスタするためには，データ・ブックを隅々まで読んで，できるだけ多くのゲートの特性を覚えることをお勧めします．

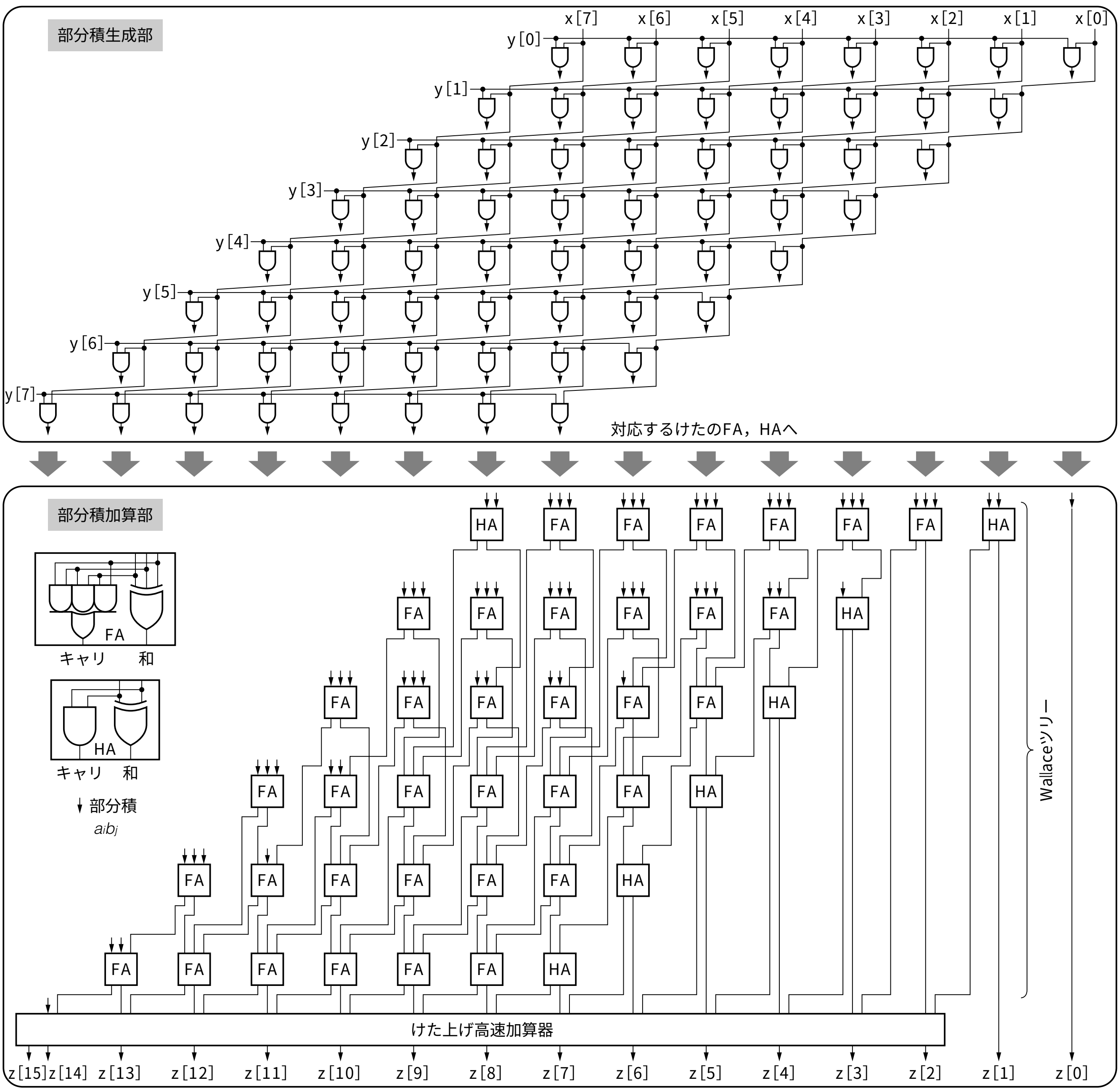


図7　8ビット×8ビット整数乗算器の構成

Wallaceツリーによってけた位置を合わせて部分積生成部からの出力を足していくときは，キャリの伝播を隣のけたで止める冗長2進形式にしておき，最終段の高速けた上げ加算器で一気にキャリをMSBまで伝播させる．

2 GF(2^n)演算はAND-XOR論理の最適化

本誌では，エラー訂正符号や暗号などの実装(1)において，ガロア体GF(2^n)の演算回路が何度か取り上げられています．整数演算の加算器は，下位ビットで発生したキャリをどのように最上位ビットまで伝播させるかが高速化のかぎとなります．これに対して，GF(2^n)上の加算はビットごとのXORなのでキャリが発生せず，ビット数に関係なくクリティカル・パスはXOR1段分となり，非常に高速です．

図7と**図8**に，8ビット整数乗算器とGF(2^8)乗算器の例を示します．整数乗算器は，Wallaceツリー部でn個(nビット)の部分積を3入力2出力のフル・アダー約$\log_{3/2}n$段で足し合わせ，最後にキャリ・ルックアヘッド加算器などの高速けた上げ加算器でキャリをMSB(most significant bit)まで伝播させます．このWallaceツリー部は不規則なうえ，フル・アダーの遅延時間は入出力間で異なるため，ビット数が増えるに従って最適化が難しくなります．一方，GF(2^n)の部分積加算はけた上がりがなく，付加回路の剰余演算部を含めても非常に簡単な構成になります．このため単体のGF(2^n)加算器や乗算器では最適化の余地はあまりありませんが，**図9**に示した暗号回路(1)などでよく使われる乗法の逆元を求める回路にはトランジスタ・レベルの回路構成を考慮した高速・小型化のテクニックが利用できます．

GF(2^n)の演算は，**図8**や**図9**に示したように一般に

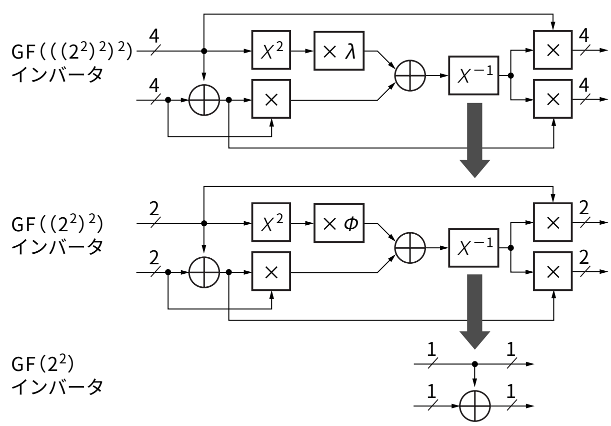


図9　合成体を用いたGF逆元回路
合成体GF(((2^2)^2)^2)の階層構造を利用した小型GF逆元回路も，ほかのガロア体の演算回路と同様にANDとXORゲートだけで構成される．

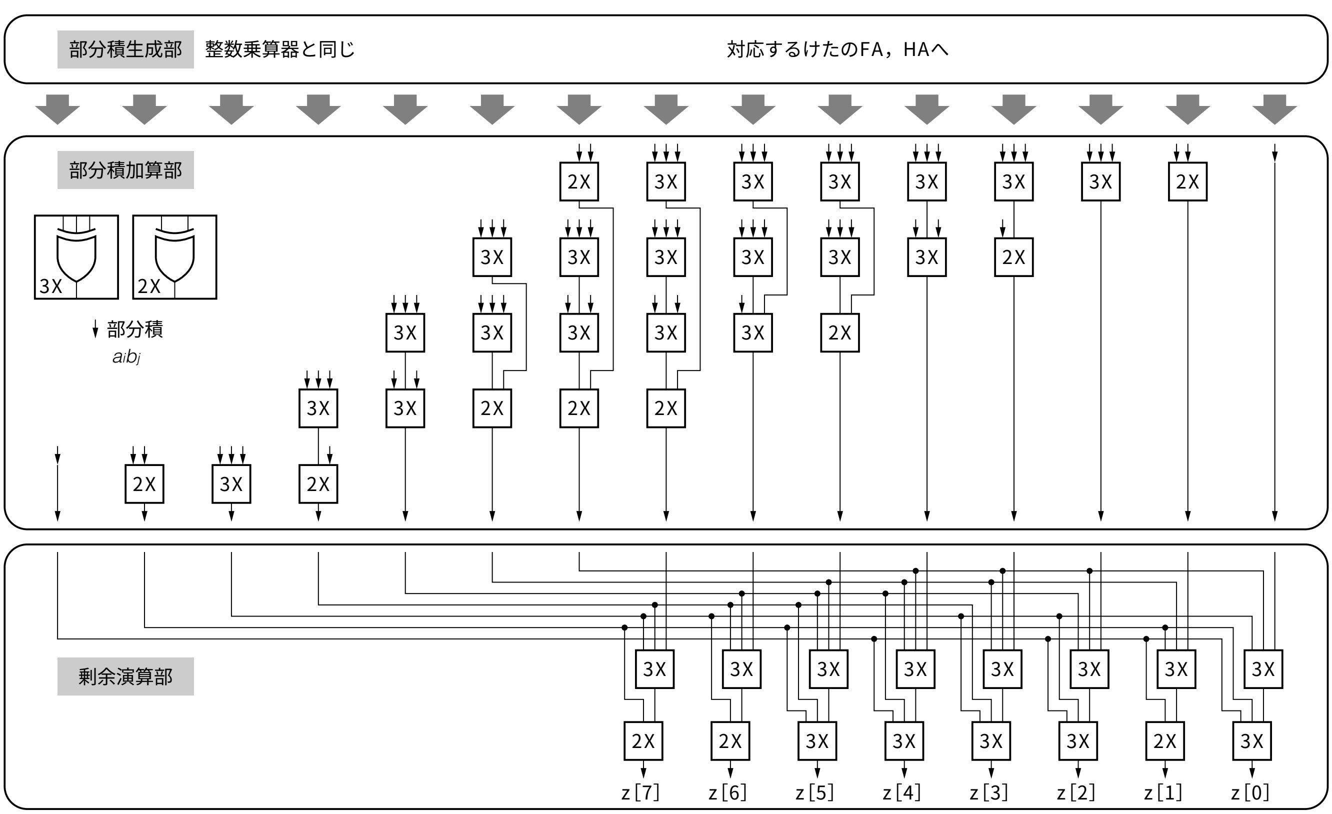
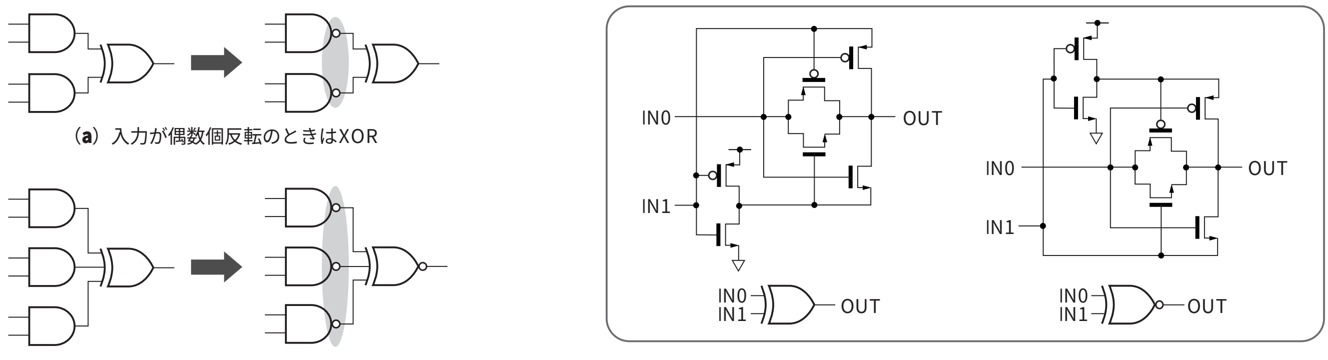


図8　GF(2^8)乗算器の構成(規約多項式 $g(x)=x^8+x^4+x^3+x^2+1$)
整数乗算器はキャリの伝播をどうするかが高速化のかぎであったが，ガロア体の加算にはけた上げがないため，非常に高速かつシンプルな乗算器となる．



（**a**）入力が偶数個反転のときはXOR

（**b**）奇数個反転のときはXNOR

（**c**）XORとXNORの回路コストは同じ

図10　AND-XOR論理の変換
XORとXNORの性能は同等なので，AND-XOR論理は入力信号の数に応じてNAND-XORあるいはNAND-XNORに変換すると小型で高速な回路となる．

AND-XOR論理で記述されます．CMOS回路では正論理と負論理を入れ替えながら論理を組んでいくと，段数の少ないスマートな回路を作ることができることを先に述べました．では，AND-XOR論理は単純にNAND-XNOR論理に置き換えればよいかというと，そうではありません．2入力XORは，入力が両方とも反転すると出力は反転しないからです．そのため，**図10**のようにNAND-XOR論理となります．また，3入力XORに対する三つのANDをNANDで置き換えた場合は，XORをXNORにすることになります．

ところで**図9**の逆元回路で，各信号はAND-XOR-AND-XORとそろって次段に伝達するわけではなく，長短さまざまな経路を通過しています．したがって，信号を反転させていくと，かならずしも極性がそろった状態で各ゲートに入力されるとは限らず，それを合わせようとすると余計なインバータなどが必要となるように思えます．しかし幸いなことに，一般的なXORとXNORはいずれも**図10(c)**に示したように最小構成が6トランジスタであり，動作速度もほとんど差はありません．ですから，XORとXNORをうまく使い分けることで簡単に極性を合わせることができるのです．

● トランジスタの抵抗を考慮した多入力ゲート選び

NAND-XORやNAND-XNORで入力の極性を反転して，NOR-XORやNOR-XNORに置き換えても，ゲート段数やトランジスタ数に変わりはありません．しかし，動作速度の点では前者のほうが有利です．

図5で3入力NANDゲートとNORゲートのトランジスタ構成を示しましたが，入力信号の数が増えるたびに，NANDではnMOSトランジスタが，NORではpMOSトランジスタがシリアルに積み上げられていきます．同じ大きさのトランジスタでは大ざっぱに言ってpMOSはnMOSの約3倍の抵抗を持つので，このpMOSを積み上げるNORゲートの使用は高速化の点で不利となります．シリコン上のレイアウトでは抵抗を下げるためにpMOSトランジスタはnMOSよりも大きく設計されていますが，シリコン面積だけでなく入力容量なども増加してしまうので，nMOSに対して2倍以下のサイズを用いることが一般的です．したがって，NANDとNORに限らず，pMOSを積み上げた多入力ゲート（もちろん内部構成がわかればの話だが…）を避けた論理式の展開を行うことが，高速化と小型化の双方で重要となります．

次に多入力XORの構成を見てみましょう．通常の3入力XORや4入力XORは，**図11**のように2入力XORを単純に複数並べた構造になっています．では，2入力XORを複数使うのと変わりがないかというと，そうではありません．二つの2入力XORを使った場合，それらは配置配線ツールによって独立にシリコン上に配置され，ゲート間の入出力端子が金属配線で接続されることになります．これに対して，初めから3入力XORとしてレイアウトされたものは1ヵ所にかためてトランジスタが置かれるため，余計な配線による抵抗や容量そして面積の増加がない分，高速で小型になるのです．

XORに限らず，トランジスタ構成が同一であってもレイアウト・レベルで見ると性能が大きく異なる場合があり，また使用できる複合ゲートの種類もライブラリによってまちまちです．ですから，それらのゲートを臨機応変に使い分けて，さまざまな論理展開を行える力を身に付けてください．

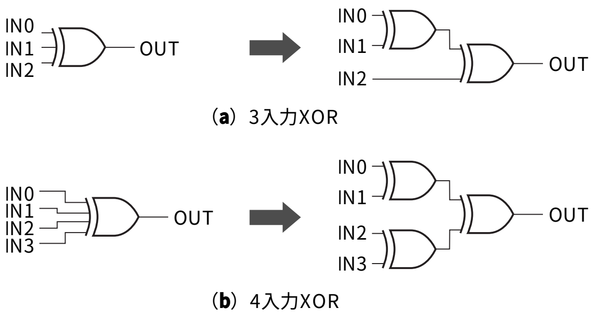


図11　多入力XORゲート
通常の3入力XORや4入力XORは，2入力XORを単純に複数並べた構造になっている．

表1　SubByte関数（8ビット入出力）の性能比較
ゲート数は2入力NAND換算，クリティカル・パスの遅延時間はワースト・ケースの値である．

	ゲート数	遅延（ns）	AT積	比　率
論理合成ツール	295	3.45	1,017.8	1.000
手マップ	287	3.07	881.1	0.866

表2　AES回路全体の性能比較

	ゲート数	遅延（ns）	AT積	比　率
論理合成ツール	8,045	8.85	71,224.8	1.000
手マップ	5,398	7.62	41,132.8	0.578

3　手マップの実力

手マップによる回路最適化はレジスタ間の論理が深いもの（論理段数が大きい）ほど効果が高いのですが，その場合は勘に頼ったり力技になることも少なくありません．また，手マップの初心者はあまり効果が上がらず，努力が報われないと感じることがあるかもしれません．そこで筆者が暗号回路に対して手マップを行った例を紹介しながら，その有効性を示しましょう．

表1にSubBytes関数だけの回路について，また**表2**にそれを使ったAES（advanced encryption standard）回路について，0.13μmのCMOSスタンダード・セル・ライブラリを用いて米国Synopsys社のDesign Compilerで論理合成した場合と，手マップした場合の性能比較を示します．ゲート数は2入力NAND換算で，クリティカル・パスの遅延時間はワースト・ケースの値です．AT積はゲート数と遅延時間の積（Area × Time）なので，この値が小さければ小さいほど，小型で高速な優れた実装であると言えます．SubBytes関数は論理合成の前に，**図9**のGF逆元演算回路を用いた最適化を行っており，回路規模も小さいため，手マップによる効果は13%の改善とあまり大きいとは言えません．これに対して，AES全体ではSubBytes以外の関数もクリティカル・パスに入って論理が深くなるので，手マップによる最適化の余地が広がり，42%という大幅な性能向上となりました．このように論理合成ツールは局所的な最適化は得意なため，小さな回路では手マップに迫る性能が得られます．一方，回路規模が大きくなるとやはり全体を見渡せる人間（の勘？）にはまだまだ及ばないようです．

パソコンの飛躍的な性能向上，そして設計手法とツールの進歩によって，ソフトウェアとハードウェアの開発者の間の垣根がなくなりつつあります．そのような中，ディスプレイ上のシミュレーション波形やツールが出力するタイミング・レポートだけを見て安心してしまう人も多いのではないでしょうか．しかし，その設計データがマスク・パターンに変換され，シリコン上にアナログ特性を持つトランジスタとして形成され，そこに電流が流れて初めてLSIとして動作することを忘れないでください．

演算回路の動作を論理式ではなく，トランジスタのON/OFFとしてとらえられれば，手マップによってツールには不可能なレベルの回路最適化が可能となります．手マップは論理ゲートという多数のピースを，高性能回路というフレームにうまく当てはめるジグソー・パズルのようなものかもしれません．しかし，その解はひと通りではなく，無限の可能性がありますので，みなさんも独自の手マップ技の開発に挑戦してみてください．

参考・引用*文献

（1）* Dr. S. Morioka；数学いらずのAES暗号SubBytes設計ガイド，Design Wave Magazine，pp.152-156，2004年1月号．

さとう・あかし
日本アイ・ビー・エム（株）　東京基礎研究所

<筆者プロフィール>
佐藤証．苦行の末に，論理式を見るとマスク・パターンが目の前に浮かんでくる特殊能力を身に付けた私ですが，次はマスク・パターンを論理式に展開する技に挑戦しようかと思案中．LSIのリバース・エンジニアリングでは重要なんですよ．人手ではあまりやりませんが…．
akashi@jp.ibm.com

6